# Une méthode de conception de programmes fonctionnels

Raymond Durand, Martine Vergne

IRIA

CENTRE DE RENNES
IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 494

# UNE MÉTHODE DE CONCEPTION DE PROGRAMMES FONCTIONNELS

Raymond DURAND
Martine VERGNE

Mars 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

*Publication Interne n° 279*
*16 pages*
*Janvier 1986*

# UNE MÉTHODE DE CONCEPTION DE PROGRAMMES FONCTIONNELS

Raymond DURAND (*)

Martine VERGNE (**)

**RESUME** : On décrit sur un exemple une méthode de construction de programmes fonctionnels, suivie d'une transformation en une forme quasi-itérative. Un système d'aide au développement de programme est ensuite présentée.

**ABSTRACT** : A method to construct functionnal programs is described, followed by a transformation into a quasi-iterative form. Then, we present an interactive system of program development.

(*) LIF, Université PARIS 6
(**) IRISA, Campus de Beaulieu, 35042 RENNES CEDEX.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

## Introduction

Confrontés depuis plusieurs années à l'enseignement de la programmation, nous avons développé une méthodologie de construction de programmes fonctionnels. En effet, si de nombreux travaux théoriques, pratiques et didactiques nous servent de référence [1, 2, 11, 12, 16, 20, 21, 25] pour l'enseignement de la programmation impérative, la programmation fonctionnelle a suscité de nombreuses recherches théoriques mais peu de travaux pédagogiques à notre connaissance [3].

Chacun acquiert une expérience qui incite à utiliser de préférence ce que l'on connait le mieux (fonctionnel, impératif, déclaratif,...). Nous pensons que certains problèmes se résolvent mieux en fonctionnel qu'en impératif, mais lorsque nous échouons, nous essayons une autre méthode, le plus important étant de trouver une solution (en espérant une bonne solution).

Nos spécifications utilisent pour l'instant le langage logique du premier ordre (mais nous n'avons pas encore développé ce point difficile d'un langage de spécification) ; remarquons que cette spécification peut parfois être considérée comme un programme de type déclaratif. Par souci d'efficacité, le programme obtenu d'après les spécifications par une méthodologie qui fait l'objet de cet article est souvent transformé pour être meilleur (en place, en temps,... au choix de l'utilisateur). Enfin notre langage est conçu pour être compilé et non interprété.

Nous allons tout d'abord décrire très succintement le langage utilisé. Puis nous expliquerons les différents points de notre démarche en l'illustrant par un exemple :

- spécification du problème,
- construction du programme,
- transformation,
- implantation.

Nous décrirons ensuite notre projet à long terme d'un logiciel d'aide au développement de programme, dont la partie système de transformation est en cours d'élaboration.

## 1°) Description du langage

Notre langage fonctionnel comprend le langage de description des graphes de fonctions et le langage de description des types. Il n'est pas indispensable d'avoir deux catégories d'objets [15, 24], mais nous pensons que c'est préférable pour des raisons méthodologiques et pour la lisibilité des programmes [4, 13].

Les graphes des fonctions sont définis de façon classique directement ou récursivement à partir de fonctions "de base" et de deux opérateurs : la composition des fonctions (notation parenthésée) et l'alternative (notation "si ... alors ... sinon").

Dans le langage des types, nous notons l'ensemble et le type de la même

manière. Classiquement, un langage de types comprend deux constructeurs : le produit cartésien et l'exponentiation. Comme nous éprouvons le besoin de mieux préciser les types afin de pouvoir effectuer des controles statiques, nous avons deux constructeurs ($\Sigma$ et $\prod$) généralisant les constructeurs classiques :

$\sum_{i:I} X_i = \{x / \exists i\ x \in X_i\} = \bigcup_{i:I} X_i$ avec comme fonctions de base :

- indice de $\bigcup_{i \in I} X_i$ dans I
- Pour tout i de I : $p\ell_i$ (pour plongement) de $X_i$ dans $\bigcup_{i \in I} X_i$

  $res_i$ (pour restriction) de $\bigcup_{i \in I} X_i$ dans $X_i$

telles que : i) Pour tout x de $\bigcup_{i \in I} X_i$, x appartient à $X_{indice(x)}$

ii) $p\ell_i$ associe, à tout élément de $X_i$, lui même

iii) pour tout x de $\bigcup_{i \in I} X_i$ : • si $x \in X_i$, $res_i(x) = x$

• si $x \notin X_i$, $res_i(x)$ n'est pas défini.

$\prod_{i:I} X_i = \{f / f:I \to \bigcup_{i \in I} X_i$ et $\forall i \in I\ f(i)\downarrow \supset f(i) \in X_i\}$

($f(i)\downarrow$ signifie que f est définie en i)

(Remarquons que "$\prod$" généralise les constructeurs "produit cartésien" et "exponentiation")

Intuitivement, ce constructeur peut, par exemple, servir à décrire le type d'un programme calculant la transposée d'une matrice de dimension quelconque, la première donnée étant la dimension et la deuxième une matrice carrée de cette dimension : on notera $\prod_{n:\mathbb{N}}(([1..n] \to \mathbb{N}) \to ([1..n] \to \mathbb{N}))$ le type de la fonction "transposition".

2°) Le problème et sa spécification

Illustrons notre démarche par un exemple : de nombreux problèmes concrets mènent après formalisation à la recherche de la fermeture transitive (FT) d'un graphe ou d'une relation. La spécification de ce problème peut être :

Soit X un ensemble (l'ensemble des sommets du graphe G).

Soit U l'ensemble des arcs (qui est un sous ensemble de $X^2$). Sa fermeture transitive $FT_X(U)$ est définie par :

$(x_0, x_{n+1}) \in FT_X(U)$ ssi $\exists n \in \mathbb{N}\ \exists x_1,\ldots,x_n \in X\ \forall i\ (0<i<n)\ (x_i, x_{i+1}) \in U$

avec $FT_X : \mathcal{P}(X^2) \to \mathcal{P}(X^2)$.

*Spécifier un problème est l'étape peut-être la plus difficile. Dans notre exemple, le problème étant déja abstrait, la spécification en est relativement aisée. La difficulté réside en l'abstraction d'un problème concret.*

3°) La construction du programme

Notre choix est de construire un programme fonctionnel (et non impératif, rappelons qu'ici la spécification peut être considérée comme un programme déclaratif).

a) – Examiner la spécification

Dans notre exemple, on peut soit faire une récurrence sur "$\mathbb{N}$" ($\exists n \in \mathbb{N}$), soit faire une récurrence sur "X" ($\exists x_1,\ldots,x_n \in X$).

Faire varier "$\mathbb{N}$" nous conduirait à l'algorithme construisant la fermeture transitive en calculant les puissances successives de la matrice de représentation du graphe. Faire varier "X" va nous conduire à l'algorithme, plus performant, de Roy-Warshall-Nolin (RWN). Bien sûr à l'examen des spécifications il est difficile, pour un novice, de faire le bon choix (ici faire varier "X"). On peut cependant remarquer que le "$\mathbb{N}$" ne fait pas à proprement parler du problème qui pourait s'écrire "il existe une suite (finie) ..."; mais nous ne sommes plus alors dans le premier ordre. Cette remarque peut inciter à faire le bon choix.

Faire varier "X" conduit à la spécification d'un problème plus général que $FT_X$ que nous appelerons RWN (pour Roy-Warshall-Nolin) :

$RWN : \mathcal{P}(X)\times\mathcal{P}(X^2)\rightarrow\mathcal{P}(X^2)$

$(x_0, x_{n+1}) \in RWN(Z,U)$ ssi $\exists n\in\mathbb{N}\ \exists x_1,\ldots,x_n\in Z\ \forall i\ (0<i<n)\ (x_i, x_{i+1})\in U$

on a alors $FT_X(U) = RWN(X,U)$

*Nous venons de voir dans l'exemple une heuristique pour résoudre un problème. Il en existe bien sûr beaucoup d'autres : nous en verrons une autre à la fin de ce paragraphe et on peut aussi penser aux nombreux problèmes de recherche opérationnelle où il faut trouver une solution vérifiant certains critères, alors qu'il est plus facile d'écrire un programme donnant toutes les solutions vérifiant ces critères.*

*Remarquons que ces heuristiques sont du genre :*

*étant donné un problème ou plus exactement une spécification d'un problème :*

- *résoudre un (des) autre(s) problème(s),*
- *et le problème posé est alors fonction de ce(s) nouveau(x) problème(s).*

*Dans notre projet de logiciel, un sous-système "**Ecriture**" doit aider à les trouver.*

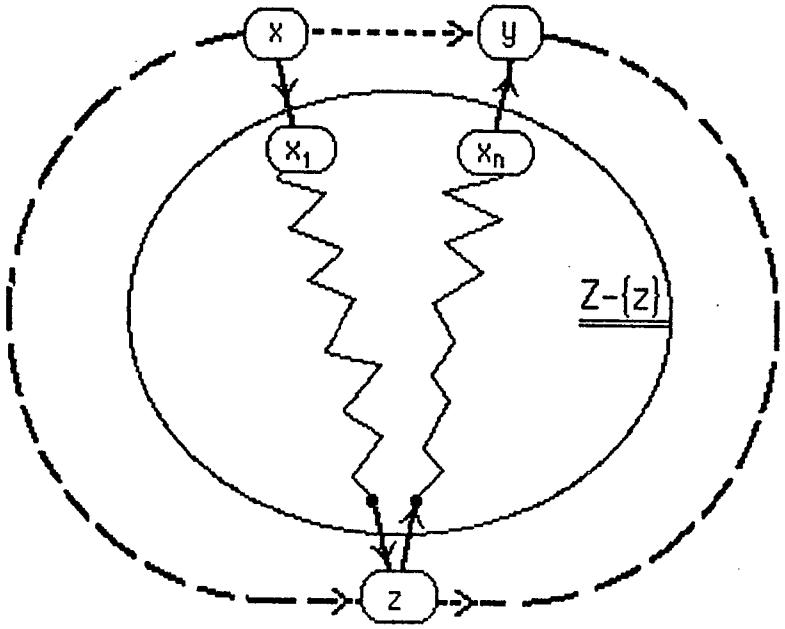
b) Trouver une relation de récurrence

Nous devons chercher une relation permettant de résoudre RWN(Z,U) connaissant la solution de RWN(α(Z,U)) où α(Z,U) est "plus petit" que (Z,U) (dans cet exemple, "U" ne varie pas et on remplacera α(Z,U) par (α(Z), U) ). Cherchons les arcs à "rajouter" à RWN(α(Z),U) (on remarque que l'on peut se limiter aux chemins élémentaires) : considérons z un élément de Z et (x,y) appartenant à RWN(Z,U) et alors soient $x_1,...,x_n$ des éléments de Z tels que

$\forall i\ (1<i<n)\ (x_i,x_{i+1}) \in U$ et $(x, x_1) \in U$ et $(x_n,y) \in U$. Deux cas peuvent se présenter :

1°) si $z \notin \{x_1,...,x_n\}$ (z n'est pas sur le chemin), $(x,y) \in RWN(Z-\{z\},U)$

2°) si $z \in \{x_1,...,x_n\}$,




----- Arc à rajouter dans RWN(Z,U)

—— Arcs appartenant à RWN(Z-{z},U)

(x,z) et (z,y) appartiennent alors à RWN(Z-{z},U) par définition de ce dernier.
En résumé : pour tout élément z de Z, RWN(Z,U) est égal à
RWN(Z-{z},U) ∪ {(x,y) / (x,z) ∈ RWN(Z-{z},U) & (z,y) ∈ RWN(Z-{z},U)}.
On remarque alors que "U" ne varie pas dans cette relation et il peut (et même doit) donc être considéré comme une variable globale, ce que l'on peut écrire dans notre langage :

$RWN_U(Z) = RWN_U(Z-\{z\}) \cup \{(x,y) / (x,z) \in RWN_U(Z-\{z\}) \ \&\ (z,y) \in RWN_U(Z-\{z\})\}$

(où $RWN_U$ doit être considéré comme un identificateur de fonction).

ou encore : pour tout élément z de Z,

(1) $RWN_U(Z) = \theta(z, RWN_U(Z-\{z\}))$

% $\theta : X \times \mathcal{P}(X^2) \rightarrow \mathcal{P}(X^2)$

% $\theta(z,V) = V \cup \{(x,y) / (x,z) \in V \ \& \ (z,y) \in V\}$

$RWN_U(Z)$ étant, pour tout z de Z, égal à $\theta(z, RWN_U(Z-\{z\}))$, est en particulier, égal à $\theta(c(Z), RWN_U(Z-\{c(Z)\}))$ où c : $\lceil U : \mathcal{P}(X) \rceil U$ est une fonction de choix, ce qui constitue bien une relation de récurrence.

c) Vérifier que la récurrence "s'arrête" :
Pour cela, on choisit un ordre bien fondé $<$ sur $\mathcal{P}(X)$ tel qu'en général $\alpha(Z)<Z$. Il suffit de prendre ici l'inclusion ensembliste.

d) Enlever les valeurs (par si...alors...sinon...) pour lesquelles soit $\alpha$ n'est pas définie, soit $\alpha(Z)$ n'est pas inférieur à Z. Dans notre cas $Z-\{c(Z)\}$ n'est pas défini pour $Z=\emptyset$ et on remarque alors que $RWN_U(\emptyset)$ est égal à U. On obtient finalement la définition :

(2) $RWN_U(Z) \Leftarrow$ Si $Z=\emptyset$
alors U
sinon $\theta(c(Z), RWN_U(Z-\{c(Z)\}))$

Résumons la méthodologie employée dans le cas général :
Soit f:X→Y l'application à calculer,
- on commence par trouver une "relation de récurrence", c'est-à-dire une expression $\theta(\varepsilon x, f\alpha_0 x, \ldots, f\alpha_n x)$ telle, qu'en général, fx soit égal à $\theta(\varepsilon x, f\alpha_0 x, \ldots, f\alpha_n x)$
- on choisit un ordre bien fondé $<$ sur X tel qu'en général $\alpha_0 x < x, \alpha_1 x < x, \ldots, \alpha_n x < x$
- on "enlève" (par un ou plusieurs si alors sinon) les valeurs x pour lesquelles :
  - soit $\alpha_0 x$ ou $\alpha_n x$ ou $\varepsilon x$ ne sont pas définis
  - soit non $\alpha x < x$
  - soit $f(x) \neq \theta(\varepsilon x, f\alpha_0 x, \ldots, f\alpha_n x)$
- si l'on connait fx pour ces valeurs, on a terminé, sinon il faut chercher une nouvelle relation de récurrence pour ces valeurs, affiner si besoin l'ordre bien fondé et recommencer.

*Dans le logiciel, le système "**Ecriture**" aidera interactivement l'usager en lui suggérant les différentes étapes à effectuer, les vérifications et contrôles pouvant être effectués directement par le système (est-ce un ordre bien fondé etc...)*

Comme autre suggestion possible on peut conseiller à l'utilisateur la proposition suivante :

($\mathcal{P}_f(G)$ est l'ensemble des parties finies de G)

Soit $\underline{f} : \mathcal{P}_f(G) \to Y$ une fonction telle que

$$\forall X \in G \; \forall x \in X \; \underline{f}(X) = \theta(x, \underline{f}(X-\{x\}))$$
$$\underline{f}(\emptyset) = a$$

$\underline{f}$ est calculée par le programme :

F(X,a)

$F : \mathcal{P}_f(G) \times Y \to Y$

$F(X,U) \Leftarrow$ Si $X=\emptyset$
alors U
sinon $F(X-\{c(X)\}, \theta(c(X),U))$

% c : $\lceil_{U : \mathcal{P}(X)}\rceil U$ (fonction de choix)

En utilisant cette proposition sur la relation (1) ci-dessus, et en remarquant que $RWN_U(\emptyset)$ est égal à U (qui est bien une constante pour $RWN_U$), on obtient la définition de la fermeture transitive :

$FT_X(U) = RWN(X,U) = RWN_U(X) = RWNI(X,U)$

(3) $RWNI : \mathcal{P}(X) \times \mathcal{P}(X^2) \to \mathcal{P}(X^2)$

$RWNI(Z,V) \Leftarrow$ Si $Z=\emptyset$
alors V
sinon $RWNI(Z-\{c(Z)\}, \theta(c(Z),V))$

Cette définition est quasi-itérative.

*Après cette étape, la définition obtenue est juste mais n'est peut-être pas sous une forme performante. Nous allons effectuer des transformations.*

4°) Transformations [5, 6, 17, 18]

Nous ne transformerons pas la définition (3), mais uniquement la (2).

Proposition (G. Huet & B. Lang et R.B. Kieburtz & J. Schultis)

$f(x) \Leftarrow$ Si $\pi x$
alors $\varphi x$
sinon $\theta(\beta x, f\alpha x)$

$f,\varphi$ : $X \to Y$
$\pi$ : $X \to$ {vrai,faux}
$\alpha$ : $X \to X$
$\beta$ : $X \to Z$
$\theta$ : $Z \times Y \to Y$ qui soit stricte par rapport au 2° argument

Soit G telle que : (G est une fonction qui "descend")
$G : X \to Y$
$G(x) \Leftarrow$ Si $\pi x$
alors $\varphi x$
sinon $G\alpha x$
soit $\theta^* : Y \times Z \to Y$ telle que :
(1) $\theta(z,\theta^*(y,z_1)) = \theta^*(\theta(z,y),z_1)$
(on dit que $\theta$ et $\theta^*$ sont associatives duales)

(2) $\theta(\beta x, Gx) = \theta^*(Gx,\beta x)$ (on dit que G est un pivot pour $\theta,\theta^*$)

Soit K définie par : (K est la fonction qui "remonte")
$K : X \times Y \to Y$
$K(x,y) \Leftarrow$ Si $\pi x$
alors y
sinon $K(\alpha x,\theta^*(y,\beta x))$
Alors
$f(x)=K(x,G(x))$

On peut vérifier (grâce à des propriétés sur le "et" et le "ou" logique) que
$\forall x \forall y \forall V\ \theta(x,\theta(y,V)) = \theta(y,\theta(x,V))$,
$\theta$ et $\theta^*$, égale à $\lambda Vz.\theta(z,V)$, sont donc associatives duales et toute fonction G, en particulier $\lambda z.U$, est un pivot pour $\theta$, $\theta^*$.
Le schéma précédent nous conduit à la définition :
$FT_X(U) = RWN(X,U) = RWN_U(X) = RWNI(X,U)$

(4) $RWNI : \mathcal{P}(X) \times \mathcal{P}(X^2) \to \mathcal{P}(X^2)$
$RWNI(Z,V) \Leftarrow$ Si $Z=\emptyset$
alors V
sinon $RWNI(Z-\{c(Z)\}, \theta(c(Z),V))$
On retrouve d'une autre manière la définition (3).

*Il existe un grand nombre de schémas. Notre préoccupation est de trouver des schémas très généraux dégageant certains concepts (par exemple associativité duale, quasi-associativité, généralisation de la notion de pile,...) que nous étudions afin de*

*pouvoir utiliser un système expert minimisant le rôle des "eureka" de Darlington par exemple. Cette partie* ***"Transformation"*** *du système doit pouvoir tenir compte de certains vœux de l'utilisateur (amélioration en temps ou en mémoire)*

5°) Implantation

Nous allons compiler cette définition quasi-itérative en utilisant l'inverse de la transformation de Mac-Carthy. Pour la clarté, nous traduirons dans un langage genre Pascal. A ce stade nous devons choisir les structures pour l'implémentation des types abstraits (comment représenter X et U, i.e. les sommets et les arcs). L'idée courante est de numéroter les sommets et d'utiliser une matrice booléenne pour les arcs. Ainsi, X est représenté par un intervalle d'entier : c1..c2 ; si z est un entier de cet intervalle, il représente l'intervalle Z égal à (z..c2), la valeur de la fonction de choix en Z est alors z et Z-[c(Z)] est l'intervalle z+1..c2 représenté par z+1; U est représenté par une matrice [c1..c2, c1..c2] de booléens et V = θ(z,V) se traduit par :

```
pour x←c1 jusqu'à c2 faire
  pour y←c1 jusqu'à c2 faire
        U[x,y]←U[x,y] ou (U[x,z] et U[z,y])
  Fait
Fait
```

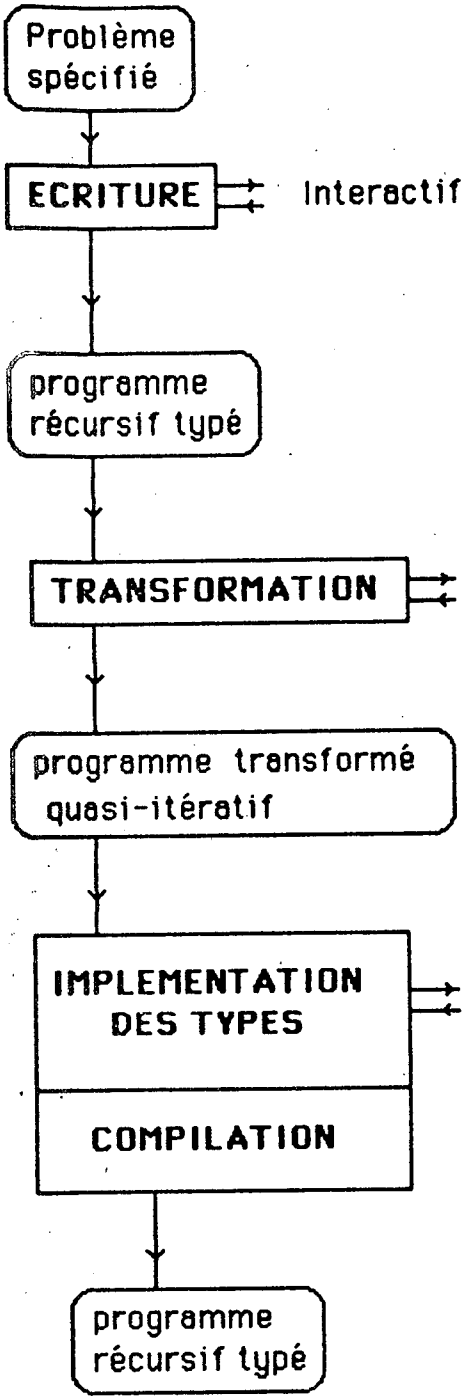
ou mieux par :

```
pour x←c1 jusqu'à c2 faire
  Si U[x,z] alors pour y←c1 jusqu'à c2 faire U[x,y]←U[x,y] ou U[z,y] fait fsi
fait
```

*La grosse difficulté se trouve dans la partie implémentation des types. Les choix peuvent être guidés par les fonctions de base : on choisira de préférence des structures facilitant leur calcul. Nous pensons utiliser une solution système expert. La partie compilation présente moins de difficultés; nous envisageons de compiler en P-code (comme en pascal-ucsd) pour des raisons de transportabilité. Nous allons maintenant décrire notre projet à long terme d'un logiciel aidant le programme à effectuer les 5 points précédents.*

6°) Le système

Le système complet se présente sous la forme suivante :




ECRITURE aide à trouver une définition fonctionnelle typée, en indiquant des "trucs", en faisant ou en suggérant des vérifications, en guidant la méthodologie.
Cela correspond au §3.

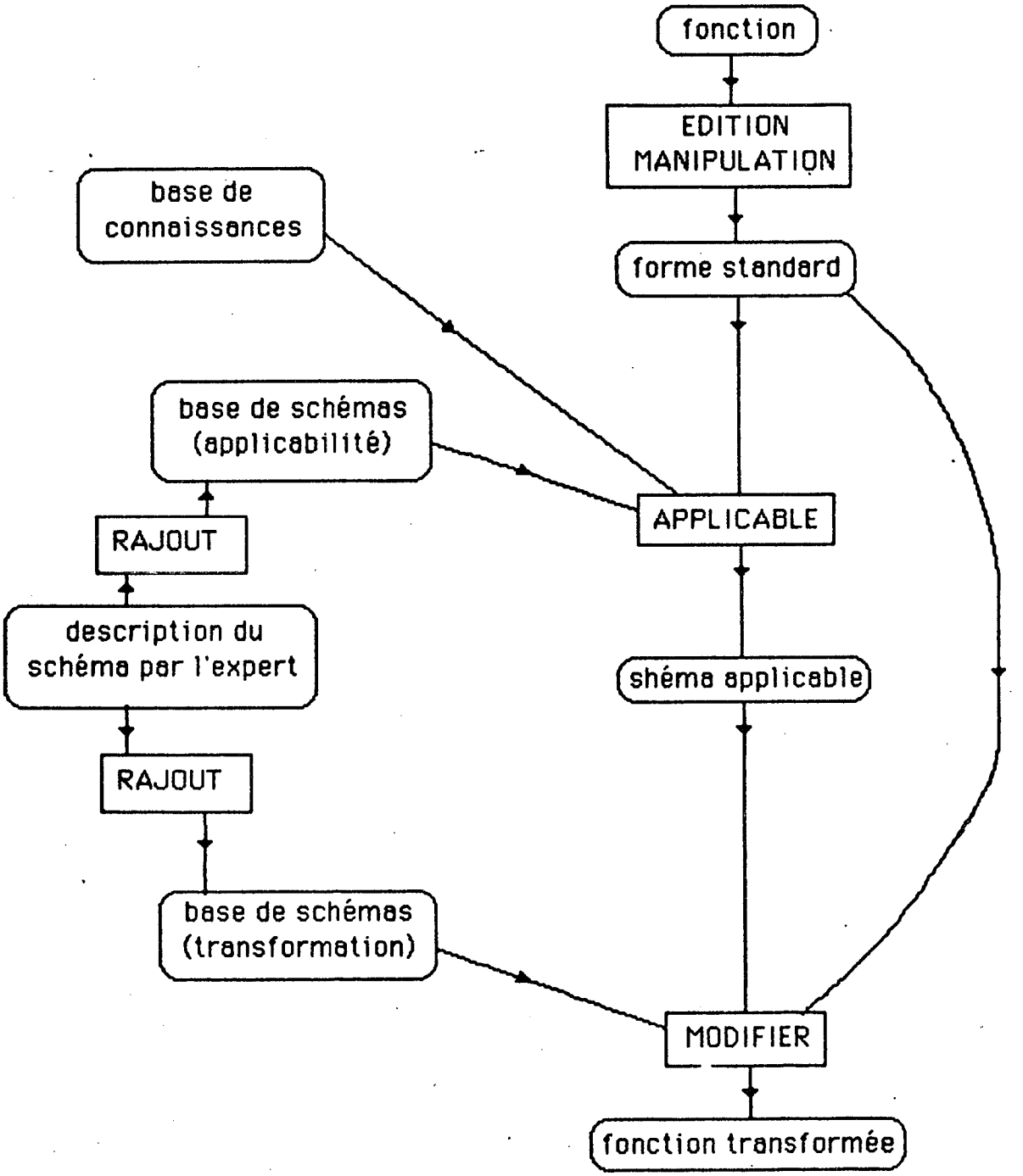
TRANSFORMATION donne un programme transformé selon les vœux de l'utilisateur. Cela correspond au §4.

IMPLEMENTATION & COMPILATION correspond au §5

Nous avons commencé l'implémentation de la partie TRANSFORMATION. Le système est organisé autour d'une base de connaissance des schémas de transformation et d'une base de connaissance de propriétés des fonctions soit couramment utilisées (monde universel), soit utilisées spécifiquement (monde utilisateur) soit acquises pour la circonstance (monde actuel).
La description (faite par l'expert) de chaque schéma comporte
* une description syntaxique.
* une description sémantique des propriétés que doivent posséder les diverses fonctions du schéma.

* une note de l'intérêt du schéma vis à vis d'un ensemble de critères par exemple gain en temps, gain en mémoire, nombreuses contraintes à examiner, etc.
L'expert doit pouvoir rajouter des schémas, modifier les critères d'intérêt afin que lui-même (et non le système ni l'utilisateur) soit responsable de l'ordre du choix des schémas.

DESCRIPTION DU SYSTEME DE TRANSFORMATION



EDITION MANIPULATION : la donnée est un programme syntaxiquement correct construit rappelons-le par le système ECRITURE d'aide à l'écriture d'un programme fonctionnel. Cette définition fait en général appel à d'autres fonctions (puisque notre méthodologie descendante découpe le problème en plusieurs sous problèmes). On sait qu'une récursivité croisée peut toujours se ramener à une récursivité simple en codant les fonctions à l'aide de n-uplets. Mais certaines simplifications sont plus performantes : soit par exemple la fonction f qui appelle g. Si g n'appelle pas f, il suffit de recopier la définition de g dans f pour n'avoir plus qu'un seul appel. Ce problème de savoir quelles recopies sont possibles revient à effectuer un tri topologique sur le graphe d'appel des fonctions, à reconnaitre les circuits, à déterminer quels sont les noeuds équivalents (au sens : pas de réduction possible). Ces problèmes sont traités par EDITION MANIPULATION dont le résultat est une fonction sous une représentation standard.
APPLICABLE détermine le schéma applicable. Les recherches sont conduites par la forme de la fonction (appel monadique, diadique..) et ses propriétés; il fait appel à un système expert capable de trouver dans la base de connaissances ou de déduire les propriétés de fonctions qui sont nécessaires. Comme aux autres niveaux, il existe une grande interactivité. Rappelons que l'utilisateur émet des vœux en rapport avec les critères notés des schémas (gain de temps,...). Parmi les schémas applicables, on choisit celui qui a la plus forte note.
MODIFIER effectue la transformation.
RAJOUT se situe à un autre niveau dans le système. Il est activé lorsque l'expert rajoute un schéma. Il y a alors mise à jour de la base des schémas.
A tout niveau, il y a possibilité d'interaction. De plus, on peut rentrer dans le système à différents endroits. Par exemple, un utilisateur plus averti peut entrer dans le système avec une fonction déjà sous forme standard ou bien indiquer le schéma applicable. Nous avons insisté sur les transformations récursif en itératif, mais d'autres transformations sont possibles. Parmi elles, certaines que nous appelons simplifications sont systématiquement appliquées : suppression de variable inefficaces, utilisation de variables globales... A chaque transformation, ces simplifications peuvent être mises à jour.

CONCLUSION

Au moment de l'écriture de l'article, une partie de APPLICABLE et de MODIFIER sont écrites et en cours de tests (avec une base de schéma et de fait très réduites). Les résultats sont encourageants. Bien sûr nous ne pouvons prétendre être déjà au niveau de système tel que HOPE [7, 8, 9, 10], OBJ2 [13] ou ML[14]. Nous espérons arriver à obtenir des transformations plus générales, mais surtout à intégrer cette partie dans un système d'aide au développement de programmes. Par ailleurs, nous avons testé notre

méthodologie avec des étudiants de licence d'informatique à l'Université Paris 6. Les résultats sont inégaux; ils sont en général bon pour les étudiants n'ayant aucune connaissance de programmation alors que les étudiants ayant déjà eu une initiation (au basic par exemple...) sont réfractaires à l'idée de récurrence (mais ce sont souvent les mêmes qui sont réfractaires à une programmation structurée qu'elle soit impérative ou fonctionnelle). Nous ne croyons pas à un style universel de programamtion, mais nous essayons d'aider à l'apprentissage d'un style possible. Dans ce sens, nous suivons le même cheminement que Bauer [13].

Bibliographie


[1] ARSAC J. Les bases de la programmation - Paris 1983 -(Dunod Informatique)

[2] De BAKKER J. W Least fixed points revisited, Proc. of Symposium on λ-Calculus and Computer Science Theory, Rome, 1975.

[3] BAUER F.L. Program Development by Stepwise Transformation- the project CIP, International Summer School on Program Construction, Eds. F.L. Bauer and M. Broy, Lecture Notes in Computer Science, n° 69, 1979.

[4] BERRY G. Programming with concret data structures and sequential algorithms. Functionnal programming languages and computer architecture, A.C.M. proceedings of the 1981 conference Porsmouth New Hampshire.

[5] BIRD R. Notes on recursion elimination, Comm. ACM, Vol. 20, n°6, June 1977.

[6] BIRD R. Tabulation techniques for recursive programs, Computing Surveys, Vol. 12, n°4, Dec. 1980.

[7] BURSTALL R.M. et DARLINGTON J. A transformation system for developing recursive programs, J.A.C.M. 24, Jan 1976.

[8] BURSTALL R.M., MACQUEEN D.B. et SANELLA D.T.
Hope : an experimental applicative language
Internal Report, CSR 62-80 University of Edinburgh, May 1980.

[9] DARLINGTON J. A semantic approach to automatic program improvement,
Ph D thesis Univ. of Edinburgh, Report EP27, 1972.

[10] DARLINGTON J. et BURSTALL R.M.
A system which automatically improves programs,
Acta Informatica 6,1, 1976.

[11] DURAND R. Conception de programmes applicatifs : méthodologie et transformations.
Thèse d'état, Université Paris VI, 1985.

[12] FLOYD R. Assigning meanings to programs,
Proc. Symposium in applied Mathematics, J.T. SCHWARTZ,
Ed. American Mathematical Society, Mathematical aspects of Computer Science, n°19, 1967.

[13] FUTATSUGI K., GAUGEN J., JOUANNAUD J.P., MESEGUER J.
Principles of OBJ2,
Proc. of ACM Symp. on Principles of programming languages, 1984.

[14] GORDON M., MILNER R., WADSWORTH C.
Edinburgh LCF,
Lecture Notes in Computer Science n°78, Springer Verlag, 1979.

[15] GUTTAG J.V.,HORNING J.
The algebraic specification of abstract data type,
Acta Informatica vol 10 n°1, 1978.

[16] HOARE C. An axiomatic basis of computer programming,
Com. ACM, Vol. 12, n°10, Oct. 1969.

[17] HUET & LANG Proving and applying program transformations expressed with second order patterns,
Acta Informatica 11, 1978.

[18] KIEBURTZ R.B. and SCHULTIS J. :
Transformation of F.P. program Schemes,
Functionnal programming languages and computer architecture ACM proceedings of the 1981 conference Porsmouth, New Hampshire.

[19] KOTT L.
Des substitutions dans les systèmes d'équations algébriques sur le magma. Application aux transformations de programmes et à leur correction,
Thèse d'Etat, Univ. Paris VIII, 1979.

[20] KNUTH D.
The art of computer programming,
Vol. 1, 1968, Vol. 3, 1973.
Addison-Wesley, New-York.

[21] LIVERCY C.
Théorie des programmes (schémas, preuves, sémantique),
Dunod Informatique, 1978.

[22] Mac CARTHY J.
Recursive Functions of Symbolic Expressions and their Computation by Machine Part 1,
Comm. ACM, Vol. 3, n°4, April 1960.

[23] MANNA Z.
A Mathematical Theory of computation.
Mac Graw Hill computer science series 1974.

[24] PAIR C.
Types abstraits et sémantique algébrique des langages de programmation,
CRIN Nancy, 80 R 011, 1980.

[25] RAYMOND F.H.
Informatique-Programmation
Masson-EAP 1980

[26] SCHOLL P.C.
Algorithmique et représentation des données,
Vol. 3, Récursivité et arbres,
Masson, 1984.

PI 272 **Architecture pour les opérations geométriques en synthèse d'images par facettes**
François Charot, Franck Rousée – 24 pages ; Novembre 85.

PI 273 **Madmacs : a new VLSI layout macro editor**
Patrice Frison, Eric Gautrin – 12 pages ; Novembre 85.

PI 274 **Détection de pannes et reconfiguration automatique**
Michèle Basseville – 26 pages ; Novembre 85.

PI 275 **Estimation de l'ordre d'un processus Arma à l'aide de résultats de perturbations de matrices**
Jean-Jacques Fuchs – 40 pages ; Décembre 85.

PI 276 **Detection and diagnosis of changes in the eigenstructure of nonstationary multivariable systems**
Michèle Basseville, Albert Benveniste, Georges Moustakides, Anne Rougée – 44 pages ; Décembre 85.

PI 277 **Optimum Robust Detection of Changes in the Ar Part of a Multivariable Process**
Michèle Basseville, Anne Rougée, Georges Moustakides, Albert Benveniste – 48 pages ; Décembre 85.

PI 278 **Controlling knowledge transfers in distributed algorithms. Application to deadlock detection**
Jean-Michel Hélary, Aomar Maddi, Michel Raynal – 32 pages ; Janvier 1986.

PI 279 **Une méthode de conception de programmes fonctionnels**
Raymond Durand, Martine Vergne – 16 pages ; Janvier 1986.

Raymond DURAND

Martine VERGNE

# UNE METHODE DE CONCEPTION DE PROGRAMMES FONCTIONNELS

Publication interne
n° 279

Janvier 1986

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique